Theory of Computation, Unit 1: Introduction and Regular Languages

Franklin and Marshall College

August 25, 2025

Introduction

What is this class about?

We're basically trying to understand computation

1. What does it mean to compute something?

Introduction

What is this class about?

We're basically trying to understand computation

1. What does it mean to compute something?

Introduction

What is this class about?
We're basically trying to understand computation

1. What does it mean to compute something? We'll get several different answers to this question, corresponding to different types of machines or "automata". This is *automata theory*.

Introduction

What is this class about?
We're basically trying to understand computation

1. What does it mean to compute something? We'll get several different answers to this question, corresponding to different types of machines or "automata". This is *automata theory*.
2. For a given way of understanding computation, how powerful is it? I.e. what kinds of questions can it answer? This is *computability theory*

Introduction

What is this class about?

We're basically trying to understand computation

1. What does it mean to compute something? We'll get several different answers to this question, corresponding to different types of machines or "automata". This is *automata theory*.
2. For a given way of understanding computation, how powerful is it? I.e. what kinds of questions can it answer? This is *computability theory*

Introduction

What is this class about?

We're basically trying to understand computation

1. What does it mean to compute something? We'll get several different answers to this question, corresponding to different types of machines or "automata". This is *automata theory*.
2. For a given way of understanding computation, how powerful is it? I.e. what kinds of questions can it answer? This is *computability theory*
3. For a given way of understanding computation and a particular problem, how hard is it to solve? This is *complexity theory*.

Computation as Parsing

We can formalize the types of questions we'll be asking as follows:

1. Let A be some set of strings over some alphabet (e..g binary strings)
2. For a given type of machine (automaton), is there a machine $M_A(w)$ which can take a string w as input and determine where or not $w \in A$.
3. If there is such a machine, how complicated is M_A

For example:

- A is the set of all binary strings which end in 0. Is $0110 \in A$? Is $0111 \in A$

Computation as Parsing

We can formalize the types of questions we'll be asking as follows:

1. Let A be some set of strings over some alphabet (e..g binary strings)
2. For a given type of machine (automaton), is there a machine $M_A(w)$ which can take a string w as input and determine where or not $w \in A$.
3. If there is such a machine, how complicated is M_A

For example:

- A is the set of all binary strings which end in 0. Is $0110 \in A$? Is $0111 \in A$
- A is the set of all strings made out of (and) which are properly nested.

Computation as Parsing

We can formalize the types of questions we'll be asking as follows:

1. Let A be some set of strings over some alphabet (e..g binary strings)
2. For a given type of machine (automaton), is there a machine $M_A(w)$ which can take a string w as input and determine where or not $w \in A$.
3. If there is such a machine, how complicated is M_A

For example:

- A is the set of all binary strings which end in 0. Is $0110 \in A$? Is $0111 \in A$
- A is the set of all strings made out of (and) which are properly nested.
- A is the set of all strings which form a Python program which halts in 20 steps.

Computation as Parsing

What kind of formalism do we need to do syntax highlighting?

```
def fib(n):
    if n==0:
        return 0
    elif n==1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

print([fib(i) for i in range(12)])
```

Introduction

Why should I care?

1. You often have to choose some kind of formal language to work with a given system. It is important to understand the tradeoffs.
   Examples:
   1. Using a description logic in semantic web
   2. Developing a query language to work with a database.
   3. Developing a scripting language to work with an application.

Introduction

Why should I care?

1. You often have to choose some kind of formal language to work with a given system. It is important to understand the tradeoffs.
2. The mathematical tools we develop here will be useful in other contexts.
   Examples:
   1. Regular expressions are basic programming constructs; good regexp parsers work by building DFAs and NFAs.
   2. Grammars are frequently used to describe programming languages and are the basis for building compilers.

Computation

We'll study three classical models of computation: finite automata, context-free grammars and Turing machines. We can think of these of models of computation with finite memory, stack-based memory and random access memory.

Computation

We'll study three classical models of computation: finite automata, context-free grammars and Turing machines. We can think of these of models of computation with finite memory, stack-based memory and random access memory.

Each of these will be associated with a set of problems it can solve, or language. Basically, given an infinite set of **strings** over some *alphabet*, a language will be some subset of those strings. Different machines can carve out different subsets; these will be associated "languages".

Chomsky Hierarchy

Here's a summary of what we'll learn for computability:

Machine	**Language**	**Example**
Finite Automata	Regular Languages	Finding strings which match a pattern; lexical analysis
Push-Down Automata	Context Free Languages	Parsing the syntax of a Python program
Turing Machine	Recursive Languages	Factoring an integer

Other Questions

We will study classical machines and classical languages. A number of other questions could be explored using the same approach we'll apply this semester:

1. Can a simple type of neural network known as a perceptron be trained to imitate any binary operation?

Other Questions

We will study classical machines and classical languages. A number of other questions could be explored using the same approach we'll apply this semester:

1. Can a simple type of neural network known as a perceptron be trained to imitate any binary operation?
2. Are there types of computation which ChatGPT is, in principle, incapable of answering?

Other Questions

We will study classical machines and classical languages. A number of other questions could be explored using the same approach we'll apply this semester:

1. Can a simple type of neural network known as a perceptron be trained to imitate any binary operation?
2. Are there types of computation which ChatGPT is, in principle, incapable of answering?
3. Could you ever program a computer to play the perfect games of Chess?

Other Questions

We will study classical machines and classical languages. A number of other questions could be explored using the same approach we'll apply this semester:

1. Can a simple type of neural network known as a perceptron be trained to imitate any binary operation?
2. Are there types of computation which ChatGPT is, in principle, incapable of answering?
3. Could you ever program a computer to play the perfect games of Chess?
4. Could you ever program a team of computers to play the perfect game of Mario Kart?

Other Questions

We will study classical machines and classical languages. A number of other questions could be explored using the same approach we'll apply this semester:

1. Can a simple type of neural network known as a perceptron be trained to imitate any binary operation?
2. Are there types of computation which ChatGPT is, in principle, incapable of answering?
3. Could you ever program a computer to play the perfect games of Chess?
4. Could you ever program a team of computers to play the perfect game of Mario Kart?
5. Could you devise a SQL query to return all of a person's ancestors from a geneaology database?

Other Questions

We will study classical machines and classical languages. A number of other questions could be explored using the same approach we'll apply this semester:

1. Can a simple type of neural network known as a perceptron be trained to imitate any binary operation?
2. Are there types of computation which ChatGPT is, in principle, incapable of answering?
3. Could you ever program a computer to play the perfect games of Chess?
4. Could you ever program a team of computers to play the perfect game of Mario Kart?
5. Could you devise a SQL query to return all of a person's ancestors from a geneaology database?
6. Could you ever accurately simulate a real physical system?

Other Questions

We will study classical machines and classical languages. A number of other questions could be explored using the same approach we'll apply this semester:

1. Can a simple type of neural network known as a perceptron be trained to imitate any binary operation?
2. Are there types of computation which ChatGPT is, in principle, incapable of answering?
3. Could you ever program a computer to play the perfect games of Chess?
4. Could you ever program a team of computers to play the perfect game of Mario Kart?
5. Could you devise a SQL query to return all of a person's ancestors from a geneaology database?
6. Could you ever accurately simulate a real physical system?
7. Is artificial general intelligence possible?

Why the Formality?

Our approach to studying these questions will be very abstract and formally mathematical. Why?

1. A lot of the phenomena we'll be exploring are somewhat subtle and require precise definitions.

Why the Formality?

Our approach to studying these questions will be very abstract and formally mathematical. Why?

1. A lot of the phenomena we'll be exploring are somewhat subtle and require precise definitions.
2. Mathematics is a language that accurately describes the behavior of computation in much the same way that it describes the physical universe.

MU

Working with the alphabet $\{ M, I, U \}$ let us imagine a machine that can produce strings according to the following rules (x, y repreesent any string):

1. If xI is produced, you can produce xIU
2. If Mx has been produced, you can produce Mxx
3. If $xIIIy$ has been produced, you can produce xUy
4. If $xUUy$ has been produced, you can produce xy

MU

Working with the alphabet $\{ M, I, U \}$ let us imagine a machine that can produce strings according to the following rules (x, y repreesent any string):

1. If xI is produced, you can produce xIU
2. If Mx has been produced, you can produce Mxx
3. If $xIIIy$ has been produced, you can produce xUy
4. If $xUUy$ has been produced, you can produce xy

If MI has been produced, can you produce MII?

MU

Working with the alphabet $\{M, I, U\}$ let us imagine a machine that can produce strings according to the following rules (x, y repreesent any string):

1. If xI is produced, you can produce xIU
2. If Mx has been produced, you can produce Mxx
3. If $xIIIy$ has been produced, you can produce xUy
4. If $xUUy$ has been produced, you can produce xy

If MI has been produced, can you produce $MIIIIU$?

MU

Working with the alphabet $\{ M, I, U \}$ let us imagine a machine that can produce strings according to the following rules (x, y repreesent any string):

1. If xI is produced, you can produce xIU
2. If Mx has been produced, you can produce Mxx
3. If $xIIIy$ has been produced, you can produce xUy
4. If $xUUy$ has been produced, you can produce xy

If MI has been produced, can you produce MU?

Sequences and Tuples

For A any set, a k-tuple from A is (informally) an ordered list of k elements from A.

Examples

1. If $A = \mathbb{Z}$, then $(3, 4, 7, 9)$ is a 4-tuple from A.

Sequences and Tuples

For A any set, a k-tuple from A is (informally) an ordered list of k elements from A.

Examples

1. If $A = \mathbb{Z}$, then $(3, 4, 7, 9)$ is a 4-tuple from A.
2. The tuple $('\mathtt{h}', '\mathtt{e}', '\mathtt{l}', '\mathtt{l}', '\mathtt{o}')$ is a 5-tuple from Σ, where Σ is the set of all Latin alphabet characters.

Sequences and Tuples

For A any set, a k-tuple from A is (informally) an ordered list of k elements from A.

Examples

1. If $A = \mathbb{Z}$, then $(3, 4, 7, 9)$ is a 4-tuple from A.
2. The tuple $('\mathtt{h}', '\mathtt{e}', '\mathtt{l}', '\mathtt{l}', '\mathtt{o}')$ is a 5-tuple from Σ, where Σ is the set of all Latin alphabet characters.
3. Note that tuples are native to Python! So `v = ('h', 'e', 'l', 'l', 'o')` sets `v` to be the corresponding tuple.

Sequences and Tuples

For A any set, a k-tuple from A is (informally) an ordered list of k elements from A.

Examples

1. If $A = \mathbb{Z}$, then $(3, 4, 7, 9)$ is a 4-tuple from A.
2. The tuple $('\mathtt{h}', '\mathtt{e}', '\mathtt{l}', '\mathtt{l}', '\mathtt{o}')$ is a 5-tuple from Σ, where Σ is the set of all Latin alphabet characters.
3. Note that tuples are native to Python! So `v = ('h', 'e', 'l', 'l', 'o')` sets `v` to be the corresponding tuple.

More generally, a tuple from $A \times B$ is a pair (a, b) with $a \in A, b \in B$

Sequences and Tuples

For A any set, a k-tuple from A is (informally) an ordered list of k elements from A.

Examples

1. If $A = \mathbb{Z}$, then $(3, 4, 7, 9)$ is a 4-tuple from A.
2. The tuple $('\mathtt{h}', '\mathtt{e}', '\mathtt{l}', '\mathtt{l}', '\mathtt{o}')$ is a 5-tuple from Σ, where Σ is the set of all Latin alphabet characters.
3. Note that tuples are native to Python! So `v = ('h', 'e', 'l', 'l', 'o')` sets `v` to be the corresponding tuple.

A **sequence** can be thought of (informally) as an aribtrarily long tuple (possibly infinite). Formally a sequence a on A is a function $a : \mathbb{Z} \to A$.

Sequences and Tuples

For A any set, a k-tuple from A is (informally) an ordered list of k elements from A.

Examples

1. If $A = \mathbb{Z}$, then $(3, 4, 7, 9)$ is a 4-tuple from A.
2. The tuple $('\mathtt{h}', '\mathtt{e}', '\mathtt{l}', '\mathtt{l}', '\mathtt{o}')$ is a 5-tuple from Σ, where Σ is the set of all Latin alphabet characters.
3. Note that tuples are native to Python! So `v = ('h', 'e', 'l', 'l', 'o')` sets `v` to be the corresponding tuple.

The *Cartesian power*

$$A^k = \underbrace{A \times A \ldots \times A}_{k}$$

is the set of all k-tuples from A.

Relations

If A is a set then any subset of

$$\underbrace{A \times A \ldots \times A}_{k}$$

is a k-ary **relation** on A.

Examples:

1. $<$ is a relation on $\mathbb{R}$
2. "parent of" is a relation on People
3. $\{\,(x, y) \in \mathbb{Z} \times \mathbb{Z} |\ x \bmod 7 = y \bmod 7\,\}$ is an **equivalence relation** on $\mathbb{Z}$.

Functions

A function $f : A \to B$ is a set of tuples from $A \times B$ where for every $a \in A$, there is exactly one $b \in B$ with $(a, b) \in f$.

- Intuitively, we can think of f as associating $a \in A$ with $f(a) \in B$. E.g. $f(x) = x^2$ associates 3 with 9 and -2 with 4. In Python `len(s)`: $\Sigma^* \to \mathbb{N}$ and associates, e.g., `'hello'` with 5 and `'goodbye'` with 7

Functions

A function $f : A \to B$ is a set of tuples from $A \times B$ where for every $a \in A$, there is exactly one $b \in B$ with $(a, b) \in f$.

- Intuitively, we can think of f as associating $a \in A$ with $f(a) \in B$. E.g. $f(x) = x^2$ associates 3 with 9 and -2 with 4. In Python `len(s)`: $\Sigma^* \to \mathbb{N}$ and associates, e.g., `'hello'` with 5 and `'goodbye'` with 7
- We can also think of functions as associting **keys** with **values**.

Functions

A function $f : A \to B$ is a set of tuples from $A \times B$ where for every $a \in A$, there is exactly one $b \in B$ with $(a, b) \in f$.

- Intuitively, we can think of f as associating $a \in A$ with $f(a) \in B$. E.g. $f(x) = x^2$ associates 3 with 9 and -2 with 4. In Python `len(s)`: $\Sigma^* \to \mathbb{N}$ and associates, e.g., `'hello'` with 5 and `'goodbye'` with 7
- We can also think of functions as associting **keys** with **values**.
- Functions are fundamental in Python; they can be defined using `def` or as dictionaries

Functions

A function $f : A \to B$ is a set of tuples from $A \times B$ where for every $a \in A$, there is exactly one $b \in B$ with $(a, b) \in f$.

- Intuitively, we can think of f as associating $a \in A$ with $f(a) \in B$. E.g. $f(x) = x^2$ associates 3 with 9 and -2 with 4. In Python `len(s)`: $\Sigma^* \to \mathbb{N}$ and associates, e.g., `'hello'` with 5 and `'goodbye'` with 7
- We can also think of functions as associting **keys** with **values**.
- Functions are fundamental in Python; they can be defined using `def` or as dictionaries

Functions

A function $f : A \to B$ is a set of tuples from $A \times B$ where for every $a \in A$, there is exactly one $b \in B$ with $(a, b) \in f$.

- Intuitively, we can think of f as associating $a \in A$ with $f(a) \in B$. E.g. $f(x) = x^2$ associates 3 with 9 and -2 with 4. In Python `len(s)`: $\Sigma^* \to \mathbb{N}$ and associates, e.g., `'hello'` with 5 and `'goodbye'` with 7
- We can also think of functions as associting **keys** with **values**.
- Functions are fundamental in Python; they can be defined using `def` or as dictionaries

Exercises

1. Is $\{ (2, 3), (3, 2), (4, 5), (5, 3) \}$ a function? Prove your answer.
2. Explain how the Python code `f = { 3:5, 7:9, 8:10 }` defines a function. What are the domain and range?
3. Consider
```
def f(s):
  assert type(s) == str
  return s + s
```
Explain how this defines a function and give the domain, codomain and range.

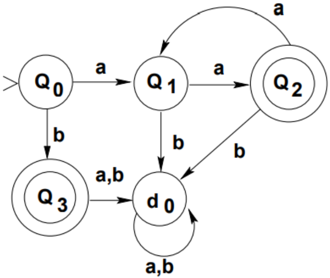
Theories and Proofs

There is a summary of proof techniques on Canvas. Try your hand at the following:

1. Let $A = \{0, 1, 2, 3, 4\}$ and let $f = \{(0, 3), (1, 1), (2, x), (3, 2), (4, 0)\}$. Prove that there is some $x \in A$ for which f is a function $A \to A$.
2. For any alphabet Σ, there is no longest element of Σ^*
3. If $a \equiv b \pmod 7$, then for any $x \in \mathbb{Z}$, $a + x \equiv b + x \pmod 7$
4. If $a \equiv b \pmod 7$, then for any $x \in \mathbb{Z}$, $ax \equiv bx \pmod 7$

Deterministic Finite Automata

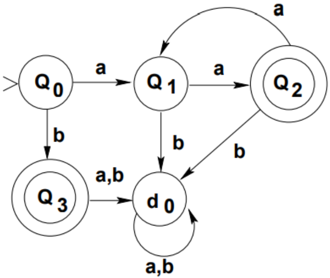
A DFA has only finite memory, represented by a finite number of states it can be in. The crucial data consists of a start state, a transition function, and accepting states.



Question: What's the starting state for this DFA?

Deterministic Finite Automata

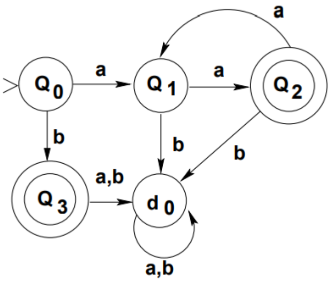
A DFA has only finite memory, represented by a finite number of states it can be in. The crucial data consists of a start state, a transition function, and accepting states.



Question: What state will the DFA be in after processing an initial "b"?

Deterministic Finite Automata

A DFA has only finite memory, represented by a finite number of states it can be in. The crucial data consists of a start state, a transition function, and accepting states.

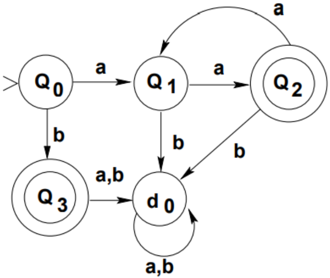


Question: What are the accepting states for this DFA?

Theory of Computation, Unit 1: Introduction and Regular Languages

Deterministic Finite Automata

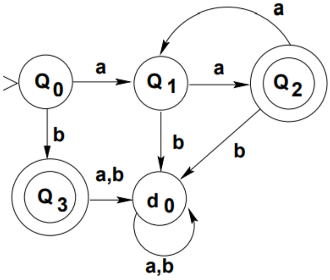
A DFA has only finite memory, represented by a finite number of states it can be in. The crucial data consists of a start state, a transition function, and accepting states.



Question: Does this DFA accept "b"?

Deterministic Finite Automata

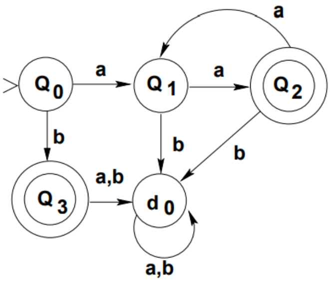
A DFA has only finite memory, represented by a finite number of states it can be in. The crucial data consists of a start state, a transition function, and accepting states.



Question: Does this DFA accept "bb"??

Deterministic Finite Automata

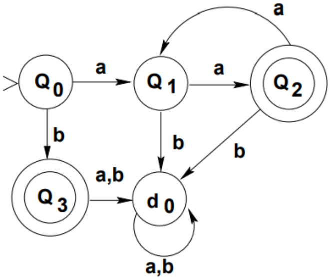
A DFA has only finite memory, represented by a finite number of states it can be in. The crucial data consists of a start state, a transition function, and accepting states.



Question: Does this DFA accept "aa"??

Deterministic Finite Automata

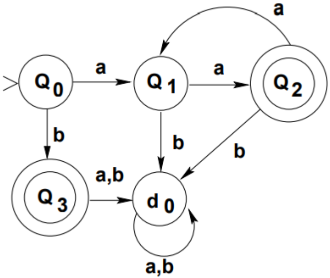
A DFA has only finite memory, represented by a finite number of states it can be in. The crucial data consists of a start state, a transition function, and accepting states.



Question: Does this DFA accept “aaa”??

Deterministic Finite Automata

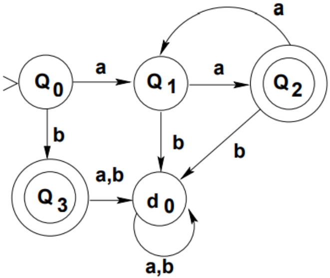
A DFA has only finite memory, represented by a finite number of states it can be in. The crucial data consists of a start state, a transition function, and accepting states.



Question: What's the accepting langauge for this DFA?

Deterministic Finite Automata

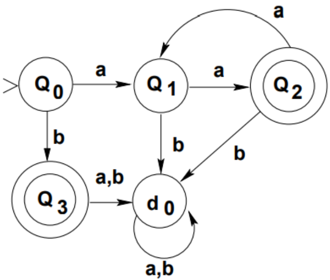
A DFA is formally a quintuple $(Q, \delta, \Sigma, q_0, F)$



where Q is a finite state of **states**. E.g.
$Q = \{\, Q_0, Q_1, Q_2, Q_3, d_0 \,\}$

Deterministic Finite Automata

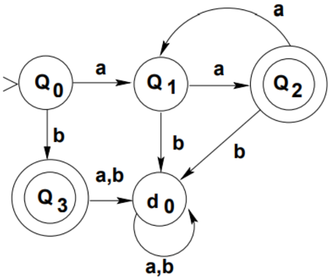
A DFA is formally a quintuple $(Q, \delta, \Sigma, q_0, F)$



where
Σ is a finite **alphabet**. E.g. $\Sigma = \{a, b\}$

Deterministic Finite Automata

A DFA is formally a quintuple $(Q, \delta, \Sigma, q_0, F)$



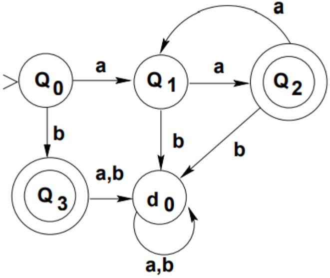
where

$\delta : Q \times \Sigma \to Q$ is the transition function. E.g.

$\delta(Q_0, a) = Q_1, \delta(Q_0, b) = Q_3, \ldots$

Deterministic Finite Automata

A DFA is formally a quintuple $(Q, \delta, \Sigma, q_0, F)$

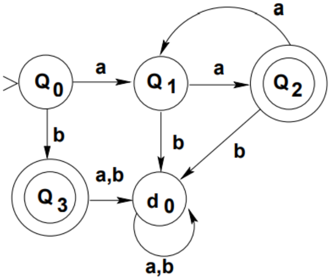


where

q_0 is the **initial state**

Deterministic Finite Automata

A DFA is formally a quintuple $(Q, \delta, \Sigma, q_0, F)$



where

$F \subseteq Q$ is the set of **accepting states**. E.g. $F = \{ Q_2, Q_3 \}$.

Deterministic Finite Automata

A DFA is formally a quintuple $(Q, \delta, \Sigma, q_0, F)$. We can think of this specification as a quintuple as specifiying 5 fields in a DFA.

```
class DFA:
    current_state = None;
    #initialize all variable when calling the class DFA
    def __init__(self, states, alphabet, transition_function, start_state, accept_states):
        self.states = states;
        self.alphabet = alphabet;
        self.transition_function = transition_function;
        self.start_state = start_state;
        self.accept_states = accept_states;
        self.current_state = start_state;

states = ['q0', 'q1']
alphabet = ['0', '1']
transitions = {
    ('q0', '0'):  'q1',
    ('q0', '1'):  'q0',
    ('q1', '0'):  'q1',
    ('q1', '1'):  'q0'
}
for (state, letter) in transitions.keys():
   assert( (state in states) and (letter in alphabet))

accept_states = ['q1']
start_state = 'q0'

d = dfa.DFA(states, alphabet, transitions, accept_states, start_state)
```

DFAs

Example

Build a DFA with alphabet $\{0, 1\}$ which recognizes binary strings that represent powers of 2.

DFAs

Example

Build a DFA with alphabet $\{0, 1\}$ which recognizes binary strings that represent odd numbers.

Theory of Computation, Unit 1: Introduction and Regular Languages

DFAs

Example

Build a DFA with alphabet $\{0, 1\}$ which recognizes binary strings with even length that end in 0.

Arithmetic Mod n

Theorem

For $n > 0$ any integer and integers a, b, c

- *Write* $a \equiv b \pmod{n}$ *when* $a \bmod n = b \bmod n$.
- *IF* $a \equiv b \pmod{n}$ *THEN* $a + c \equiv b + c \pmod{n}$
- *IF* $a \equiv b \pmod{n}$ *THEN* $ac \equiv bc \pmod{n}$

Arithmetic Mod n

Theorem

For $n > 0$ any integer and integers a, b, c

- *Write $a \equiv b \pmod{n}$ when $a \bmod n = b \bmod n$.*
- *IF $a \equiv b \pmod{n}$ THEN $a + c \equiv b + c \pmod{n}$*
- *IF $a \equiv b \pmod{n}$ THEN $ac \equiv bc \pmod{n}$*

Example

$24 \equiv 3 \pmod{7}$, so $24(5) \equiv 3(5) \pmod{7}$; i.e $24(5) \equiv 1 \pmod{7}$

Arithmetic Mod n

Theorem

For $n > 0$ any integer and integers a, b, c

- *Write $a \equiv b \pmod{n}$ when $a \bmod n = b \bmod n$.*
- *IF $a \equiv b \pmod{n}$ THEN $a + c \equiv b + c \pmod{n}$*
- *IF $a \equiv b \pmod{n}$ THEN $ac \equiv bc \pmod{n}$*

Example

$19 \equiv 5 \pmod{7}$, so $19(2) \equiv 5(2) \pmod{7}$; i.e $19(2) \equiv 3 \pmod{7}$

DFAs

Example

Build a DFA with alphabet $\{0,1\}$ which recognizes binary strings which represent multiples of 5.

MU, Revisited

Working with the alphabet $\{ M, I, U \}$ let us imagine a machine that can produce strings according to the following rules (x, y repreesent any string):

1. If xI is produced, you can produce xIU
2. If Mx has been produced, you can produce Mxx
3. If $xIIIy$ has been produced, you can produce xUy
4. If $xUUy$ has been produced, you can produce xy

MU, Revisited

Working with the alphabet $\{ M, I, U \}$ let us imagine a machine that can produce strings according to the following rules (x, y repreesent any string):

1. If xI is produced, you can produce xIU
2. If Mx has been produced, you can produce Mxx
3. If $xIIIy$ has been produced, you can produce xUy
4. If $xUUy$ has been produced, you can produce xy

Question: Can we produce "MU" from "MI"?

MU, Revisited

Working with the alphabet $\{ M, I, U \}$ let us imagine a machine that can produce strings according to the following rules (x, y repreesent any string):

1. If xI is produced, you can produce xIU
2. If Mx has been produced, you can produce Mxx
3. If $xIIIy$ has been produced, you can produce xUy
4. If $xUUy$ has been produced, you can produce xy

Question: Can we produce “MU” from “MI”?

- The only way to increase the number of Is is to double by 2).

MU, Revisited

Working with the alphabet $\{ M, I, U \}$ let us imagine a machine that can produce strings according to the following rules (x, y repreesent any string):

1. If xI is produced, you can produce xIU
2. If Mx has been produced, you can produce Mxx
3. If $xIIIy$ has been produced, you can produce xUy
4. If $xUUy$ has been produced, you can produce xy

Question: Can we produce “MU” from “MI”?

- The only way to increase the number of Is is to double by 2).
- The only way to decrease the number of Is is to subtract 3 using rule 3).

MU, Revisited

Working with the alphabet $\{M, I, U\}$ let us imagine a machine that can produce strings according to the following rules (x, y repreesent any string):

1. If xI is produced, you can produce xIU
2. If Mx has been produced, you can produce Mxx
3. If $xIIIy$ has been produced, you can produce xUy
4. If $xUUy$ has been produced, you can produce xy

Question: Can we produce "MU" from "MI"?

- The only way to increase the number of Is is to double by 2).
- The only way to decrease the number of Is is to subtract 3 using rule 3).
- Therefore, the number of Is is never divisible by 3.

MU, Revisited

Working with the alphabet $\{M, I, U\}$ let us imagine a machine that can produce strings according to the following rules (x, y repreesent any string):

1. If xI is produced, you can produce xIU
2. If Mx has been produced, you can produce Mxx
3. If $xIIIy$ has been produced, you can produce xUy
4. If $xUUy$ has been produced, you can produce xy

Question: Can we produce "MU" from "MI"?

- The only way to increase the number of Is is to double by 2).
- The only way to decrease the number of Is is to subtract 3 using rule 3).
- Therefore, the number of Is is never divisible by 3.
- Therefore you can never get rid of all the Is

Regular Languages

Definition

A language A is *regular* if there is some DFA M for which $L(M) = A$

We proved that the following are regular binary languages:

- $A = \{ w : w \text{ is the binary representation of } 2^k, k \geq 0 \}$

Regular Languages

Definition

A language A is *regular* if there is some DFA M for which $L(M) = A$

We proved that the following are regular binary languages:

- $A = \{ w : w \text{ is the binary representation of } 2^k, k \geq 0 \}$
- $A = \{ w : w \text{ is the binary representation of } 2k, k \geq 0 \}$

Regular Languages

Definition

A language A is *regular* if there is some DFA M for which $L(M) = A$

We proved that the following are regular binary languages:

- $A = \{ w : w \text{ is the binary representation of } 2^k, k \geq 0 \}$
- $A = \{ w : w \text{ is the binary representation of } 2k, k \geq 0 \}$
- $A = \{ w : |w| \text{ is even and represents } 2k, k \geq 0 \}$

Exercises

Show that each of the following is regular:

1. $E = \{\, w :$ w is the decimal representation of an even number $\}$
2. $F = \{\, w : w$ is a base 7 number that does not contain 456 as a substring $\}$
3. $G = \{\, w : w$ is the ternary representation of a number with $w \equiv 2 \pmod 5 \}$

Closure Properties

If A is regular, is that enough to know that the *complement* of A is regular? The answer is yes. Formally, we say that regular langauges are *closed under complements*.

Closure Properties

If A is regular, is that enough to know that the *complement* of A is regular? The answer is yes. Formally, we say that regular langauges are *closed under complements*.

We will show that regular languages are closed under *complements, unions, intersections* and *concatenation*.

Closure

Theorem

Let A be a regular language; then $\bar{A}$ is regular as well.

Proof.

Since A is regular, there is a DFA $M = (Q, \delta, \Sigma, q_0, F)$ which recognizes A (why?).

Closure

Theorem

Let A be a regular language; then $\bar{A}$ is regular as well.

Proof.

Since A is regular, there is a DFA $M = (Q, \delta, \Sigma, q_0, F)$ which recognizes A . We define a new DFA $M' =$

Closure

Theorem

Let A be a regular language; then $\bar{A}$ is regular as well.

Proof.

Since A is regular, there is a DFA $M = (Q, \delta, \Sigma, q_0, F)$ which recognizes A . We define a new DFA $M' = (Q, \delta, \Sigma, q_0, Q - F)$. Then we claim that M' recognizes $\bar{A}$; that is $L(M') = \bar{A}$.

Closure

Theorem

Let A be a regular language; then $\bar{A}$ is regular as well.

Proof.

Since A is regular, there is a DFA $M = (Q, \delta, \Sigma, q_0, F)$ which recognizes A . We define a new DFA $M' = (Q, \delta, \Sigma, q_0, Q - F)$. Then we claim that M' recognizes $\bar{A}$; that is $L(M') = \bar{A}$. We have to show, for $s \in \Sigma^*$, that

Closure

Theorem

Let A be a regular language; then $\bar{A}$ is regular as well.

Proof.

Since A is regular, there is a DFA $M = (Q, \delta, \Sigma, q_0, F)$ which recognizes A . We define a new DFA $M' = (Q, \delta, \Sigma, q_0, Q - F)$. Then we claim that M' recognizes $\bar{A}$; that is $L(M') = \bar{A}$. We have to show, for $s \in \Sigma^*$, that $s \in \bar{A}$ IFF $s \in L(M')$.

Closure

Theorem

Let A be a regular language; then $\bar{A}$ is regular as well.

Proof.

Since A is regular, there is a DFA $M = (Q, \delta, \Sigma, q_0, F)$ which recognizes A . We define a new DFA $M' = (Q, \delta, \Sigma, q_0, Q - F)$. Then we claim that M' recognizes $\bar{A}$; that is $L(M') = \bar{A}$. We have to show, for $s \in \Sigma^*$, that $s \in \bar{A}$ IFF $s \in L(M')$. If $s \in \bar{A}$, then M rejects s

Closure

Theorem

Let A be a regular language; then $\bar{A}$ is regular as well.

Proof.

Since A is regular, there is a DFA $M = (Q, \delta, \Sigma, q_0, F)$ which recognizes A . We define a new DFA $M' = (Q, \delta, \Sigma, q_0, Q - F)$. Then we claim that M' recognizes $\bar{A}$; that is $L(M') = \bar{A}$. We have to show, for $s \in \Sigma^*$, that $s \in \bar{A}$ IFF $s \in L(M')$. If $s \in \bar{A}$, then M rejects s ;i.e. M is in a non-accepting state after processing s.

Closure

Theorem

Let A be a regular language; then $\bar{A}$ is regular as well.

Proof.

Since A is regular, there is a DFA $M = (Q, \delta, \Sigma, q_0, F)$ which recognizes A . We define a new DFA $M' = (Q, \delta, \Sigma, q_0, Q - F)$. Then we claim that M' recognizes $\bar{A}$; that is $L(M') = \bar{A}$. We have to show, for $s \in \Sigma^*$, that $s \in \bar{A}$ IFF $s \in L(M')$. If $s \in \bar{A}$, then M rejects s . Thus M' accepts s.

Closure

Theorem

Let A be a regular language; then $\bar{A}$ is regular as well.

Proof.

Since A is regular, there is a DFA $M = (Q, \delta, \Sigma, q_0, F)$ which recognizes A . We define a new DFA $M' = (Q, \delta, \Sigma, q_0, Q - F)$. Then we claim that M' recognizes $\bar{A}$; that is $L(M') = \bar{A}$.
We have to show, for $s \in \Sigma^*$, that $s \in \bar{A}$ IFF $s \in L(M')$. If $s \in \bar{A}$, then M rejects s . Thus M' accepts s.
If $s \notin \bar{A}$, then $s \in A$ (Why?). Thus M accepts s, and M' is in a non-accepting state after processing s. □

Closure Under Intersections

Theorem

If A_1, A_2 are regular languages, then so is $A_1 \cap A_2$.

Proof.

We need to show that

Closure Under Intersections

Theorem

If A_1, A_2 are regular languages, then so is $A_1 \cap A_2$.

Proof.

We need to show that there is a DFA M which recognizes $A_1 \cap A_2$. We may assume that there are DFAs $M_1 = (Q_1, \delta_1, \Sigma, q_1, F_1), M_2 = (Q_2, \delta_2, \Sigma, q_2, F_2)$ which recognize A_1, A_2 (Why?).

Closure Under Intersections

Theorem

If A_1, A_2 are regular languages, then so is $A_1 \cap A_2$.

Proof.

We need to show that there is a DFA M which recognizes $A_1 \cap A_2$. We may assume that there are DFAs $M_1 = (Q_1, \delta_1, \Sigma, q_1, F_1), M_2 = (Q_2, \delta_2, \Sigma, q_2, F_2)$ which recognize A_1, A_2 (Why?). We want to have $M = (Q, \delta, \Sigma, q_0, F)$ simulate both machines in parallel. We define

- $Q =$

Closure Under Intersections

Theorem

If A_1, A_2 are regular languages, then so is $A_1 \cap A_2$.

Proof.

We need to show that there is a DFA M which recognizes $A_1 \cap A_2$. We may assume that there are DFAs $M_1 = (Q_1, \delta_1, \Sigma, q_1, F_1), M_2 = (Q_2, \delta_2, \Sigma, q_2, F_2)$ which recognize A_1, A_2 (Why?). We want to have $M = (Q, \delta, \Sigma, q_0, F)$ simulate both machines in parallel. We define

- $Q = Q_1 \times Q_2$

Closure Under Intersections

Theorem

If A_1, A_2 are regular languages, then so is $A_1 \cap A_2$.

Proof.

We need to show that there is a DFA M which recognizes $A_1 \cap A_2$. We may assume that there are DFAs $M_1 = (Q_1, \delta_1, \Sigma, q_1, F_1), M_2 = (Q_2, \delta_2, \Sigma, q_2, F_2)$ which recognize A_1, A_2 (Why?). We want to have $M = (Q, \delta, \Sigma, q_0, F)$ simulate both machines in parallel. We define

- $Q = Q_1 \times Q_2$
- $\delta((x, y), c) =$

Closure Under Intersections

Theorem

If A_1, A_2 are regular languages, then so is $A_1 \cap A_2$.

Proof.

We need to show that there is a DFA M which recognizes $A_1 \cap A_2$. We may assume that there are DFAs $M_1 = (Q_1, \delta_1, \Sigma, q_1, F_1), M_2 = (Q_2, \delta_2, \Sigma, q_2, F_2)$ which recognize A_1, A_2 (Why?). We want to have $M = (Q, \delta, \Sigma, q_0, F)$ simulate both machines in parallel. We define

- $Q = Q_1 \times Q_2$
- $\delta((x, y), c) = (\delta_1(x, c), \delta_2(y, c))$

Closure Under Intersections

Theorem

If A_1, A_2 are regular languages, then so is $A_1 \cap A_2$.

Proof.

We need to show that there is a DFA M which recognizes $A_1 \cap A_2$. We may assume that there are DFAs $M_1 = (Q_1, \delta_1, \Sigma, q_1, F_1), M_2 = (Q_2, \delta_2, \Sigma, q_2, F_2)$ which recognize A_1, A_2 (Why?). We want to have $M = (Q, \delta, \Sigma, q_0, F)$ simulate both machines in parallel. We define

- $Q = Q_1 \times Q_2$
- $\delta((x, y), c) = (\delta_1(x, c), \delta_2(y, c))$
- $q_0 =$

Closure Under Intersections

Theorem

If A_1, A_2 are regular languages, then so is $A_1 \cap A_2$.

Proof.

We need to show that there is a DFA M which recognizes $A_1 \cap A_2$. We may assume that there are DFAs $M_1 = (Q_1, \delta_1, \Sigma, q_1, F_1), M_2 = (Q_2, \delta_2, \Sigma, q_2, F_2)$ which recognize A_1, A_2 (Why?). We want to have $M = (Q, \delta, \Sigma, q_0, F)$ simulate both machines in parallel. We define

- $Q = Q_1 \times Q_2$
- $\delta((x, y), c) = (\delta_1(x, c), \delta_2(y, c))$
- $q_0 = (q_1, q_2)$

Closure Under Intersections

Theorem

If A_1, A_2 are regular languages, then so is $A_1 \cap A_2$.

Proof.

We need to show that there is a DFA M which recognizes $A_1 \cap A_2$. We may assume that there are DFAs $M_1 = (Q_1, \delta_1, \Sigma, q_1, F_1), M_2 = (Q_2, \delta_2, \Sigma, q_2, F_2)$ which recognize A_1, A_2 (Why?). We want to have $M = (Q, \delta, \Sigma, q_0, F)$ simulate both machines in parallel. We define

- $Q = Q_1 \times Q_2$
- $\delta((x, y), c) = (\delta_1(x, c), \delta_2(y, c))$
- $q_0 = (q_1, q_2)$
- $F =$

Closure Under Intersections

Theorem

If A_1, A_2 are regular languages, then so is $A_1 \cap A_2$.

Proof.

We need to show that there is a DFA M which recognizes $A_1 \cap A_2$. We may assume that there are DFAs $M_1 = (Q_1, \delta_1, \Sigma, q_1, F_1), M_2 = (Q_2, \delta_2, \Sigma, q_2, F_2)$ which recognize A_1, A_2 (Why?). We want to have $M = (Q, \delta, \Sigma, q_0, F)$ simulate both machines in parallel. We define

- $Q = Q_1 \times Q_2$
- $\delta((x, y), c) = (\delta_1(x, c), \delta_2(y, c))$
- $q_0 = (q_1, q_2)$
- $F = F_1 \times F_2$

Closure Under Intersections

Theorem

If A_1, A_2 are regular languages, then so is $A_1 \cap A_2$.

Proof.

We define

- $Q = Q_1 \times Q_2$
- $\delta((x, y), c) = (\delta_1(x, c), \delta_2(y, c))$
- $q_0 = (q_1, q_2)$
- $F = F_1 \times F_2$

We need to show that $s \in A_1 \cap A_2$ IFF $s \in L(M)$. If $s \in L(M)$ then M ends in state (u, v) where $u \in F_1$ and $v \in F_2$. Thus M_1 and M_2 both accept s (why?).

If $s \in A_1 \cap A_2$ then $s \in L(M)$ since the pairs of states in the transitions of M match those in M_1, M_2. □

Closure Under Intersections

Theorem

If A_1, A_2 are regular languages, then so is $A_1 \cap A_2$.

Proof.

We define

- $Q = Q_1 \times Q_2$
- $\delta((x, y), c) = (\delta_1(x, c), \delta_2(y, c))$
- $q_0 = (q_1, q_2)$
- $F = F_1 \times F_2$

We need to show that $s \in A_1 \cap A_2$ IFF $s \in L(M)$. If $s \in L(M)$ then M ends in state (u, v) where $u \in F_1$ and $v \in F_2$. Thus M_1 and M_2 both accept s (why?).

If $s \in A_1 \cap A_2$ then $s \in L(M)$ since the pairs of states in the transitions of M match those in M_1, M_2. $\square$

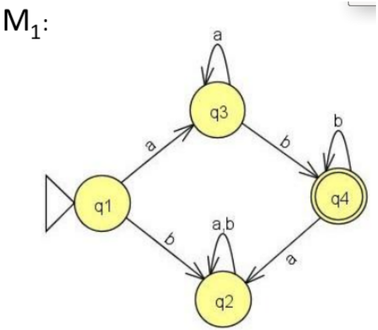
Closure Under Unions

Question

How could we modify the proof on the previous page to show that the regular languages are closed under unions?

DFA Review

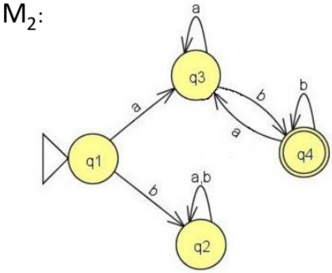
Which language is recognized by this DFA?



a) Starts with *b* and ends with *a* or *b*
b) Starts with *a* and ends with *a* or *b*
c) Some number of *a*s followed by some number of *b*s
d) Any number of *a*s followed by the same number of *b*s

DFA Review

Which language is recognized by this DFA?

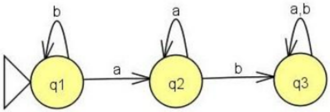


a) Starts with b and ends with b
b) Starts with a and ends with b
c) Some number of bs followed by some number of as
d) Any number of bs followed by the same number of as

DFA Review

Which set of accepting states will have M_1 recognize $\{ w : b \text{ never follows any } a \text{ in } w \}$?

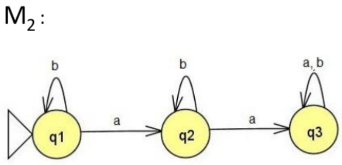
M_1:



a) $F = \{ q_2 \}$
b) $F = \{ q_3 \}$
c) $F = \{ q_1, q_2 \}$
d) $F = \{ q_1, q_3 \}$
e) $F = \{ q_2, q_3 \}$

DFA Review

Which set of accepting states will have M_2 recognize $\{ w : w \text{ does not contain exactly one } a \}$?



a) $F = \{ q_2 \}$
b) $F = \{ q_3 \}$
c) $F = \{ q_1, q_2 \}$
d) $F = \{ q_1, q_3 \}$
e) $F = \{ q_2, q_3 \}$

DFA Review

True or False: Deterministic Finite Automata (DFAs) can only recognize finite languages, not infinite languages

DFA Review

True or False: Each DFA recognizes a unique language; in other words, no two DFAs recognize the same language.

Additional Exercises

1. Let

   $$A = \{\, w : w \text{ has an odd number of } a\text{s and ends with a } b \,\}$$

   Write A as $A_1 \cap A_2$ where A_1, A_2 are simpler regular languages. Find DFAs for A_1, A_2 and use them to construct a DFA for A. Alphabet is $\{\, a, b \,\}$

2. Find a DFA for

   $$\{\, w : w \text{ contains at least two 0s and at most one 1} \,\}$$

   Alphabet is $\{\, 0, 1 \,\}$

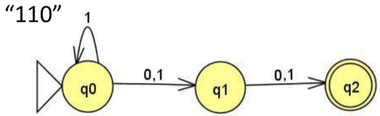
3. Find a DFA for

   $$\{\, w : w \text{ contains an even number of 0s or exactly two 1s} \,\}$$

   Alphabet is $\{\, 0, 1 \,\}$

NFA

Does the following NFA accept input 110?



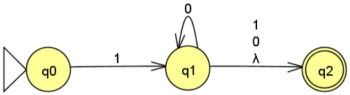
a) Yes, because some possible path ends in an accepting state.
b) Yes, because every possible path ends in an accepting state.
c) No, because some possible path ends in an rejecting state.
d) No, because every possible path ends in an rejecting state.

Theory of Computation, Unit 1: Introduction and Regular Languages

Introduction
Mathematical Preliminaries
DFAs
Regular Languages
NFAs
More Closure Properties
Regular Expressions
Non-Regular Languages

NFA

Does the following accept 100 (note that λ is used for an empty transition (ε))



a) Yes, because some possible path ends in an accepting state.
b) Yes, because every possible path ends in an accepting state.
c) No, because some possible path ends in an rejecting state.
d) No, because every possible path ends in an rejecting state.

NFA

A nondeterministic finite automaton (NFA) is formally a quintuple $(Q, \Sigma, \delta, q_0, F)$
where

- Q is a finite state of **states**.
- Σ is a finite **alphabet**. E.g. $\Sigma = \{ a, b \}$
- $\delta : Q \times \Sigma_\epsilon \to \mathbb{P}(Q)$ is the transition function.
- q_0 is the **start state**
- $F \subseteq Q$ is the set of **accepting states**.

NFA Construction

Exercises

Construct NFA to recognize the following languages with the specified number of states (all use a binary alphabet):

1. $\{ w : w \text{ ends with } 00 \}$; use 3 states.
2. $\{ w : w \text{ contains } 0101 \text{ as a substring} \}$; use 5 states.
3. $\{ w : w \text{ contains and even number of 0s or exactly two 1s} \}$; use 7 states (challenge: use 6 states).
4. $\{ 0 \}$; use 2 states.
5. $\{ 0 \}^*$; use 1 state.

Converting an NFA to a DFA (No ε transitions)

Given $N = (Q, \Sigma, \delta, q_0, F)$ define $M = (Q', \Sigma, \delta', q_0', F')$ by

- $Q' = \mathbb{P}(Q)$

Converting an NFA to a DFA (No ε transitions)

Given $N = (Q, \Sigma, \delta, q_0, F)$ define $M = (Q', \Sigma, \delta', q_0', F')$ by

- $Q' = \mathbb{P}(Q)$
- $\delta'(R, a) = \bigcup_{r \in R} \delta(r, a)$

Converting an NFA to a DFA (No ε transitions)

Given $N = (Q, \Sigma, \delta, q_0, F)$ define $M = (Q', \Sigma, \delta', q_0', F')$ by

- $Q' = \mathbb{P}(Q)$
- $\delta'(R, a) = \bigcup_{r \in R} \delta(r, a)$
- $q_0' = \{ q_0 \}$

Converting an NFA to a DFA (No ε transitions)

Given $N = (Q, \Sigma, \delta, q_0, F)$ define $M = (Q', \Sigma, \delta', q_0', F')$ by

- $Q' = \mathbb{P}(Q)$
- $\delta'(R, a) = \bigcup_{r \in R} \delta(r, a)$
- $q_0' = \{ q_0 \}$
- $F' = \{ R \in Q' : R \text{ contains an accept state of } N \}$

Converting an NFA to a DFA (With ε transitions)

Given $N = (Q, \Sigma, \delta, q_0, F)$, for $R \subseteq Q$, define

$E(R) = \{ q : q$ is reachable from R by using 0 or more ε transition

define $M = (Q', \Sigma, \delta', q_0', F')$ by

- $Q' = \mathbb{P}(Q)$

Converting an NFA to a DFA (With ε transitions)

Given $N = (Q, \Sigma, \delta, q_0, F)$, for $R \subseteq Q$, define

$E(R) = \{ q : q$ is reachable from R by using 0 or more ε transition

define $M = (Q', \Sigma, \delta', q_0', F')$ by

- $Q' = \mathbb{P}(Q)$
- $\delta'(R, a) = \bigcup_{r \in R} E(\delta(r, a))$

Converting an NFA to a DFA (With ε transitions)

Given $N = (Q, \Sigma, \delta, q_0, F)$, for $R \subseteq Q$, define

$E(R) = \{ q : q$ is reachable from R by using 0 or more ε transition

define $M = (Q', \Sigma, \delta', q_0', F')$ by

- $Q' = \mathbb{P}(Q)$
- $\delta'(R, a) = \bigcup_{r \in R} E(\delta(r, a))$
- $q_0' = E(\{ q_0 \})$

Converting an NFA to a DFA (With ε transitions)

Given $N = (Q, \Sigma, \delta, q_0, F)$, for $R \subseteq Q$, define

$E(R) = \{ q : q$ is reachable from R by using 0 or more ε transition

define $M = (Q', \Sigma, \delta', q_0', F')$ by

- $Q' = \mathbb{P}(Q)$
- $\delta'(R, a) = \bigcup_{r \in R} E(\delta(r, a))$
- $q_0' = E(\{ q_0 \})$
- $F' = \{ R \in Q' : R$ contains an accept state of $N \}$

Theory of Computation, Unit 1: Introduction and Regular Languages

Introduction
Mathematical Preliminaries
DFAs
Regular Languages
NFAs
More Closure Properties
Regular Expressions
Non-Regular Languages

Closure Under Concatenation

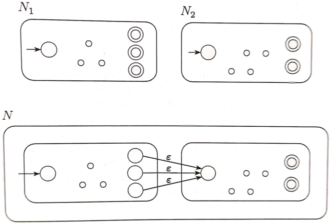
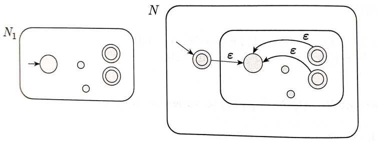


FIGURE 1.48

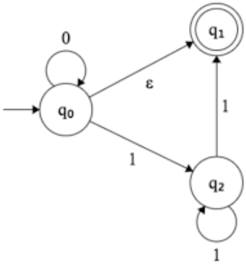
Scanned with CamScanner

Closure Under Star



Exercises

1. Construct an NFA with 3 states to recognize the language of all binary strings containing 1 as a substring and then convert it to a DFA.
2. Convert the following NFA to a DFA:



Regular Expressions

The following are regular expressions for the alphabet Σ:

- a for $a \in \Sigma$.

Regular Expressions

The following are regular expressions for the alphabet Σ:

- a for $a \in \Sigma$.
- ε

Regular Expressions

The following are regular expressions for the alphabet Σ:

- a for $a \in \Sigma$.
- ε
- $\emptyset$

Regular Expressions

The following are regular expressions for the alphabet Σ:

- a for $a \in \Sigma$.
- ε
- $\emptyset$
- $(R_1 \cup R_2)$ for R_1, R_2 regular expressions.

Regular Expressions

The following are regular expressions for the alphabet Σ:

- a for $a \in \Sigma$.
- ε
- $\emptyset$
- $(R_1 \cup R_2)$ for R_1, R_2 regular expressions.
- $(R_1 \circ R_2)$ for R_1, R_2 regular expressions.

Regular Expressions

The following are regular expressions for the alphabet Σ:

- a for $a \in \Sigma$.
- ε
- $\emptyset$
- $(R_1 \cup R_2)$ for R_1, R_2 regular expressions.
- $(R_1 \circ R_2)$ for R_1, R_2 regular expressions.
- (R_1^*) for R_1 a regular expression.

Regular Expressions

The following are regular expressions for the alphabet Σ:

- a for $a \in \Sigma$.
- ε
- $\emptyset$
- $(R_1 \cup R_2)$ for R_1, R_2 regular expressions.
- $(R_1 \circ R_2)$ for R_1, R_2 regular expressions.
- (R_1^*) for R_1 a regular expression.
- $(R_1^+) = R_1 \circ (R_1)^*$ for R_1 a regular expression.

Regular Expressions

The following are regular expressions for the alphabet Σ:

- a for $a \in \Sigma$.
- ε
- $\emptyset$
- $(R_1 \cup R_2)$ for R_1, R_2 regular expressions.
- $(R_1 \circ R_2)$ for R_1, R_2 regular expressions.
- (R_1^*) for R_1 a regular expression.
- $(R_1^+) = R_1 \circ (R_1)^*$ for R_1 a regular expression.

Regular Expressions

The following are regular expressions for the alphabet Σ:

- a for $a \in \Sigma$.
- ε
- $\emptyset$
- $(R_1 \cup R_2)$ for R_1, R_2 regular expressions.
- $(R_1 \circ R_2)$ for R_1, R_2 regular expressions.
- (R_1^*) for R_1 a regular expression.
- $(R_1^+) = R_1 \circ (R_1)^*$ for R_1 a regular expression.

Example: Find regular expressions for each of the following languages (over the alphabet $\Sigma = \{ 0, 1 \}$):

1. The set of all strings which end in 00

Regular Expressions

The following are regular expressions for the alphabet Σ:

- a for $a \in \Sigma$.
- ε
- $\emptyset$
- $(R_1 \cup R_2)$ for R_1, R_2 regular expressions.
- $(R_1 \circ R_2)$ for R_1, R_2 regular expressions.
- (R_1^*) for R_1 a regular expression.
- $(R_1^+) = R_1 \circ (R_1)^*$ for R_1 a regular expression.

Example: Find regular expressions for each of the following languages (over the alphabet $\Sigma = \{ 0, 1 \}$):

1. The set of all strings which end in 00
2. The set of all strings which contain 0101 as a substring

Regular Expressions

The following are regular expressions for the alphabet Σ:

- a for $a \in \Sigma$.
- ε
- $\emptyset$
- $(R_1 \cup R_2)$ for R_1, R_2 regular expressions.
- $(R_1 \circ R_2)$ for R_1, R_2 regular expressions.
- (R_1^*) for R_1 a regular expression.
- $(R_1^+) = R_1 \circ (R_1)^*$ for R_1 a regular expression.

Example: Find regular expressions for each of the following languages (over the alphabet $\Sigma = \{ 0, 1 \}$):

1. The set of all strings which end in 00
2. The set of all strings which contain 0101 as a substring
3. The set of all strings which contain an even number of 0s

Regular Expressions

The following are regular expressions for the alphabet Σ:

- a for $a \in \Sigma$.
- ε
- $\emptyset$
- $(R_1 \cup R_2)$ for R_1, R_2 regular expressions.
- $(R_1 \circ R_2)$ for R_1, R_2 regular expressions.
- (R_1^*) for R_1 a regular expression.
- $(R_1^+) = R_1 \circ (R_1)^*$ for R_1 a regular expression.

Example: Find regular expressions for each of the following languages (over the alphabet $\Sigma = \{ 0, 1 \}$):

1. The set of all strings which end in 00
2. The set of all strings which contain 0101 as a substring
3. The set of all strings which contain an even number of 0s
4. The set of all strings which contain an even number of 0s or exactly two 1s

Regular Expressions

The following are regular expressions for the alphabet Σ:

- a for $a \in \Sigma$.
- ε
- $\emptyset$
- $(R_1 \cup R_2)$ for R_1, R_2 regular expressions.
- $(R_1 \circ R_2)$ for R_1, R_2 regular expressions.
- (R_1^*) for R_1 a regular expression.
- $(R_1^+) = R_1 \circ (R_1)^*$ for R_1 a regular expression.

Example: Find regular expressions for each of the following languages (over the alphabet $\Sigma = \{ 0, 1 \}$):

1. The set of all strings which end in 00
2. The set of all strings which contain 0101 as a substring
3. The set of all strings which contain an even number of 0s
4. The set of all strings which contain an even number of 0s or exactly two 1s
5. The set of all strings which contain and even number of 0s and an odd number of 1s and do not contain 01 as a substring.

Theory of Computation, Unit 1: Introduction and Regular Languages

Regular Expressions

Theorem

A language is regular if and only if it is described by some regular expression.

Regular Expressions

Practice

https://regexone.com/

Please start this tutorial in class and finish at home. Some of the Problems at the end may appear in your next quiz.

Regular Expressions in Practice

If we want to see if a string w matches a regular expression r, running w through a DFA or NFA for r works in time $|w|$ – this is how the Linux program `grep` works.

Regular Expressions in Practice

If we want to see if a string w matches a regular expression r, running w through a DFA or NFA for r works in time $|w|$ – this is how the Linux program `grep` works.
Unfortunately, many programming languages use an inefficient (but more expressive) version of regular expressions by default (this is PCRE – *Perl Compatible Regular Expressions*). In Python, you can use `re2` to get fast regular expressions that are built on finite automata.

Pumping Lemma

If A is a regular language, then there is some number p such that:

if $s \in A$ with $|s| > p$, then we can write $s = xyz$ with:

1. *for* $i \geq 0$, $xy^iz \in A$
2. $|y| > 0$
3. $|xy| \leq p$

Pumping Lemma

How to use the pumping lemma to show that A is **not** regular.

- Assume, by way of contradiction, that A is regular.

Pumping Lemma

How to use the pumping lemma to show that A is **not** regular.

- Assume, by way of contradiction, that A is regular.
- Then the pumping lemma applies, so there is a "pumping length" p – any long enough $s \in A$ can be "pumped" according to the pumping lemma.

Pumping Lemma

How to use the pumping lemma to show that A is **not** regular.

- Assume, by way of contradiction, that A is regular.
- Then the pumping lemma applies, so there is a "pumping length" p – any long enough $s \in A$ can be "pumped" according to the pumping lemma.
- Choose some $s \in A$, which, when it is pumped, gives a string not in A.

Pumping Lemma

How to use the pumping lemma to show that A is **not** regular.

- Assume, by way of contradiction, that A is regular.
- Then the pumping lemma applies, so there is a "pumping length" p – any long enough $s \in A$ can be "pumped" according to the pumping lemma.
- Choose some $s \in A$, which, when it is pumped, gives a string not in A.
- This is a contradiction! So our original assumption that A was regular must be wrong.

PL

Theorem

$L = \{O^a 1^b O^a : a, b \geq 0\}$ *is not regular.*

Proof.

Assume (towards contradiction) that L is regular. Then the pumping lemma applies to L. Let p be the pumping length. Choose s to be the string ___. The pumping lemma guarantees s can be divided into parts $s = xyz$ s.t. for any $i \geq 0, xy^iz \in L$, $|y| > 0$, $|xy| < m$. But if we let $i =$ ___, we get the string $XXXX$, which is not in L, a contradiction. Therefore the assumption is false, and L is not regular. Q.E.D. $\square$

a) $s = 00000100000$, $i = 5$

b) $s = 0^p10^p$, $i = 0$

c) $s = (010)^p$, $i = 5$

d) None or more than one of the above

Pumping Lemma

How to use the pumping lemma to show that A is **not** regular.

- Assume, by way of contradiction, that A is regular.
- Then the pumping lemma applies, for $|s| > p$ with $s \in A$, s can be "pumped" according to the pumping lemma.
- Choose some $s \in A$, which, when it is pumped gives a string not in A. This is a contradiction! So our original assumption that A was regular must be wrong.

Consider

$$A = \{ 0^n 1^n : n \geq 0 \}$$

To use the pumping lemma, what's an appropriate choice of s? (Suppose p is the pumping length).

a) $s = 000111$

b) $s = 0^p 1^p 0^{2p}$

c) $s = 0^p 1^p$

d) $s = 0^{p^2} 1^p$

Pumping Lemma

How to use the pumping lemma to show that A is **not** regular.

- Assume, by way of contradiction, that A is regular.
- Then the pumping lemma applies, for $|s| > p$ with $s \in A$, s can be "pumped" according to the pumping lemma.
- Choose some $s \in A$, which, when it is pumped gives a string not in A. This is a contradiction! So our original assumption that A was regular must be wrong.

Consider

$$A = \{ 0^n 1^n : n \geq 0 \}$$

Suppose p is the pumping length. Choose $s = 0^p 1^p$. Then $s = xyz$ with $|y| > 0$, $|xy| \leq p$ and $xy^i z \in A \forall i \geq 0$. Since $|xy| \leq p$, what can we conclude about y?

a) $y = 0011$

b) $y = 0^{p-1}$

c) $|y| < |x|$

d) $y \in 0^+$

Pumping Lemma

How to use the pumping lemma to show that A is **not** regular.

- Assume, by way of contradiction, that A is regular.
- Then the pumping lemma applies, for $|s| > p$ with $s \in A$, s can be "pumped" according to the pumping lemma.
- Choose some $s \in A$, which, when it is pumped gives a string not in A. This is a contradiction! So our original assumption that A was regular must be wrong.

Consider

$$A = \{ 0^n 1^n : n \geq 0 \}$$

Suppose p is the pumping length. Choose $s = 0^p 1^p$. Then $s = xyz$ with $|y| > 0$, $|xy| \leq p$ and $xy^i z \in A \forall i \geq 0$. Since $|xy| \leq p$, we know $y = 0^k$ for some $k > 0$. Then $xy^0 z =$

a) $0^p 1^p$

b) $0^m 1^p$ where $m < p$

c) $0^k 1^p$

d) 1^p

Pumping Lemma

How to use the pumping lemma to show that A is **not** regular.

- Assume, by way of contradiction, that A is regular.
- Then the pumping lemma applies, for $|s| > p$ with $s \in A$, s can be "pumped" according to the pumping lemma.
- Choose some $s \in A$, which, when it is pumped gives a string not in A. This is a contradiction! So our original assumption that A was regular must be wrong.

Consider

$$A = \{ 0^n 1^n : n \geq 0 \}$$

Suppose p is the pumping length. Choose $s = 0^p 1^p$. Then $s = xyz$ with $|y| > 0$, $|xy| \leq p$ and $xy^i z \in A \forall i \geq 0$. Since $|xy| \leq p$, we know $y = 0^k$ for some $k > 0$. Then $xy^0 z = 0^m 1^p$ where $m < p$. But this is not in A, a contradiction!

Pumping Lemma Proofs as a 2-Player Game

Game: Given language A, Player 1 wants to use the pumping lemmma to show that A is not regular.

1. Player 1 chooses a string s; with
   1. $s \in A$
   2. $|s| \geq p$
2. Player 2 chooses x, y, z with
   1. $s = xyz$
   2. for $|y| > 0$
   3. $|xy| \leq p$
3. Player 1 wins if they can find some i so that $xy^iz \notin A$; **no matter what** choices Player 2 made.

Pumping Lemma

How to use the pumping lemma to show that A is **not** regular.

- Assume, by way of contradiction, that A is regular.
- Then the pumping lemma applies, so there is a "pumping length" p – any long enough $s \in A$ can be "pumped" according to the pumping lemma.
- Choose some $s \in A$, which, when it is pumped gives a string not in A.
- This is a contradiction! So our original assumption that A was regular must be wrong.

Show that the following binary languages are not regular.

1. $A = \{ ww^{\mathcal{R}} : w \in \Sigma^* \}$
2. $A = \{ w : n_0(w) < n_1(w) \}$
3. $A = \{ (01)^n 1^k : n > k \geq 0 \}$
4. $A = \{ 0^n : n = a^2, a \in \mathbb{Z} \}$

Pumping Lemma Proofs as a 2-Player Game

Game: Given language A, Player 1 wants to use the pumping lemmma to show that A is not regular.

1. Player 1 chooses a string s; with
   - $s \in A$
   - $|s| \geq p$
2. Player 2 chooses x, y, z with
   - $s = xyz$
   - for $|y| > 0$
   - $|xy| \leq p$
3. Player 1 wins if they can find some i so that $xy^iz \notin A$; **no matter what** choices Player 2 made.

Let's play! Choose Player 1 and Player 2; play for 3 rounds, switching players after each round. Who has a winning strategy?

Pumping Lemma Proofs as a 2-Player Game

1. Player 1 chooses a string s; with
   - $s \in A$
   - $|s| \geq p$
2. Player 2 chooses x, y, z with
   - $s = xyz$
   - for $|y| > 0$
   - $|xy| \leq p$
3. Player 1 wins if they can find some i so that $xy^iz \notin A$; **no matter what** choices Player 2 made.

Let's play! Choose Player 1 and Player 2; play for 3 rounds, switching players after each round. Who has a winning strategy?

$$\textbf{1.} \quad A = \{\, www : w \in \{\,0,1\,\}\,\}$$

Pumping Lemma Proofs as a 2-Player Game

1. Player 1 chooses a string s; with
   - $s \in A$
   - $|s| \geq p$
2. Player 2 chooses x, y, z with
   - $s = xyz$
   - for $|y| > 0$
   - $|xy| \leq p$
3. Player 1 wins if they can find some i so that $xy^iz \notin A$; **no matter what** choices Player 2 made.

Let's play! Choose Player 1 and Player 2; play for 3 rounds, switching players after each round. Who has a winning strategy?

2. $A = \{ w : w = 0^m 1^n \text{ for some natural } m, n \}$

Pumping Lemma Proofs as a 2-Player Game

1. Player 1 chooses a string s; with
   - $s \in A$
   - $|s| \geq p$
2. Player 2 chooses x, y, z with
   - $s = xyz$
   - for $|y| > 0$
   - $|xy| \leq p$
3. Player 1 wins if they can find some i so that $xy^iz \notin A$; **no matter what** choices Player 2 made.

Let's play! Choose Player 1 and Player 2; play for 3 rounds, switching players after each round. Who has a winning strategy?

$$\textbf{3.} \quad A = \{\, a^{2^n} : n \in \mathbb{N} \,\}$$

Pumping Lemma Proofs as a 2-Player Game

1. Player 1 chooses a string s; with
   - $s \in A$
   - $|s| \geq p$
2. Player 2 chooses x, y, z with
   - $s = xyz$
   - for $|y| > 0$
   - $|xy| \leq p$
3. Player 1 wins if they can find some i so that $xy^iz \notin A$; **no matter what** choices Player 2 made.

Let's play! Choose Player 1 and Player 2; play for 3 rounds, switching players after each round. Who has a winning strategy?

$$\textbf{4.} \quad A = \{ w : w \equiv 0 \pmod 3, w \in \{0, 1\} \}$$

Exam Prep

Look at problems 1.31, 1.35, 1.41, 1.47 as well as the prep problems from HW2.